Detecting speculative execution vulnerabilities on weak memory models

Nicholas Coughlin[0000−0001−8758−0666], Kait Lam[0009−0001−2599−2259], Graeme Smith(✉)[0000−0003−1019−4761], and Kirsten Winter[0000−0002−8519−2026]

Defence Science and Technology Group, Australia
School of Electrical Engineering and Computer Science,
The University of Queensland, Australia
g.smith1@uq.edu.au

Abstract. Speculative execution attacks affect all modern processors and much work has been done to develop techniques for detection of associated vulnerabilities. Modern processors also operate on weak memory models which allow out-of-order execution of code. Despite this, there is little work on looking at the interplay between speculative execution and weak memory models. In this paper, we provide an information flow logic for detecting speculative execution vulnerabilities on weak memory models. The logic is general enough to be used with any modern processor, and designed to be extensible to allow detection of vulnerabilities to specific attacks. The logic has been proven sound with respect to an abstract model of speculative execution in Isabelle/HOL.

1 Introduction

Speculative execution is a hardware optimisation in which the processor uses latent processing cycles to continue executing instructions based on a predicted value of an unevaluated expression, such as a branch condition. If the subsequent evaluation of the expression agrees with the prediction, the results of the executed instructions are committed to main memory, otherwise they are rolled back. This optimisation came to the forefront of computer security in 2018 with the disclosure of two related security attacks, Spectre [24] and Meltdown [25]. These were followed by the publication of a number of other speculative execution attacks [5,22,4,9,35,42,34], each taking advantage of traces of the speculatively executed code remaining in caches, and other micro-architectural features, after roll-back.

While much has been done to detect speculative vulnerabilities in code [8], most of this work has not considered additional hardware optimisations related to a processor's *weak memory model* [39]. All commercial processors (x86 processors of Intel and AMD, ARM processors, IBM Power, etc.) operate under a weak memory model which, again to make use of latent processing cycles, allows out-of-order execution of instructions. This out-of-order execution is constrained on individual threads so that only syntactically independent instructions may execute out-of-order, thereby guaranteeing behaviour equivalent to the original program order.

While many programmers can therefore ignore weak memory effects, those utilising data races for efficiency (e.g., programmers of low-level code of operating system routines or library components) cannot. In the presence of data races, weak memory effects can result in behaviour not apparent in the code itself. As we show in this paper, this can lead to additional speculative execution vulnerabilities. To the best of our knowledge, we are the first to show that such vulnerabilities are possible.

Early work on weak memory models and security includes that by Vaughan and Milstein [41] (for the x86 weak memory model TSO) and Mantel et al. [27] (for TSO, PSO and IBM-370). These papers highlight security violations that are not detectable using standard approaches to information flow security. In [37,38], Smith et al. provide an information flow logic for the significantly weaker memory models of ARMv8 and IBM Power processors. This approach builds on the work of Mantel et al. [28] which uses a restricted form of rely/guarantee reasoning [23,44] to allow reasoning to be done over one thread of a concurrent program at a time. The approach is adapted to more general rely/guarantee reasoning on the ARMv8 weak memory model by Coughlin and Smith [13]. While this approach has been automated using symbolic execution, its inherent complexity limits the size of the programs that can be effectively handled. In further work by Coughlin et al. [14,15], this complexity is significantly reduced via a general approach that allows standard reasoning (assuming instructions execute in program order) to be augmented with additional *reordering interference freedom* (*rif*) checks to account for the effects of a given weak memory model. As well as being simpler to apply, the approach is parameterised by the weak memory model and hence can be applied to any currently available processor.

In this paper, we adopt the *rif* approach and use it with an information flow logic developed specifically for detecting speculative execution vulnerabilities. In Section 2, we provide a brief overview of Spectre-like attacks and show via an example how weak memory effects can introduce additional speculative execution vulnerabilities. In Section 3, we provide an overview of the work on which we build: an existing information flow logic for concurrent programs [43] and the aforementioned work on *rif* to capture weak memory effects [14]. Our logic for detecting speculative execution vulnerabilities is presented and applied to our example from Section 2 in Section 4. We compare our approach to the current literature in Section 5 before concluding in Section 6.

2 Speculative Execution Attacks

Speculative execution has been liberally applied in processor design as chip-makers seek to maximise performance. As a result, there are many sources of speculation and hence many associated attacks. Canella et al. [6] taxonomise speculative execution attacks and, in doing so, reveal additional avenues for mistraining applicable to all such attacks.

Spectre attacks [24] exploit deficiencies in the process of reverting incorrect speculations. Although their primary effects are reversed, their microarchitectural side-effects are reverted incompletely or not at all. Through timing side channels, an attacker can trace these side-effects and infer information accessed during the speculation — potentially exposing sensitive information and breaking traditional software isolation. Variants of Spectre differ in the root of the speculation they exploit and also in the side channel they use to extract the information. In this paper, we focus on Spectre-PHT, variant 1 reported in the initial discovery of Spectre vulnerabilities by Kocher et al. [24].

2.1 Spectre-PHT

In Spectre-PHT, the source of speculation is a conditional branch. By mistraining the pattern history table (PHT), an attacker forces the victim code to bypass a bounds check, indexing an array out of bounds and potentially accessing sensitive information. If that information is used to index another array, the array's value at that index is loaded into the cache. Since the cache line is not reverted when speculation is cancelled, the sensitive information can be revealed by a timing difference when accessing values of the array.

The typical Spectre-PHT gadget (from [24]) is shown below. If the second line is speculatively executed when the guard in the first line is false, the value of `array1[x]` multiplied by the cache line size (4096) is used to load a value of `array2`. After roll-back, a cache timing attack will reveal the cache line which was used from which the value of `array1[x]` can be derived.

```
if (x < array1_size)
    y := array2[array1[x] * 4096]
```

2.2 Spectre-PHT and Weak Memory

As well as speculative execution, multicore processors employ pipelining and superscalar design to improve the efficiency of executed code. Several instructions are evaluated simultaneously and may take effect in an order different to their order in the program. These additional *weak memory* effects can largely be ignored when code is either not concurrent, or is concurrent but data-race free. However, these effects do need to be considered when writing efficient low-level code for device drivers and concurrent data structures. Low-level programming constructs, in particular *fences*, can be used to control instruction ordering where required.

Consider a program in which a variable `c` may hold sensitive information whenever both of the lock variables, `a` and `b`, are non-zero indicating they are held by a writing thread. Furthermore, neither `a` nor `b` change value once set to 0. In the example, we use `a`, `b` and `c` for global variables shared between threads, and `r0` to `r3` for thread-local registers. We use the notation `[r]` for dereferencing the address held in register `r`.

The reading thread below checks whether a is 0 before entering the branch. The instructions before the if statement may take time leading to speculation on the branch condition. In the branch, b and c are read and if b is 0, c's value is used as an offset from a non-sensitive base address in a subsequent read. The check of b is done using a conditional expression which is not subject to speculation. Hence, when ignoring weak memory effects, the code does not enable a Spectre-PHT attack based on the final read: when speculatively executing the branch when a is non-zero, the conditional expression evaluates to 0 when b is also non-zero.

```
r0 := a;
r1 := r1 ^ r1;      // exclusive-or sets r1 to 0
if (r0 = r1)
    r0 := b;        // this line could reorder with the next
    r1 := c;
    r2 := (r0=0) ? base + r1 : 0;
    r3 := [r2];     // Spectre attack enabled if r2 is sensitive
```

On all current microprocessors, however, the syntactically independent reads of b and c could be reordered leading to the following scenario. During speculative execution of the branch, a sensitive value held in c is read due to both a and b being non-zero. A thread in the environment then sets c to a non-sensitive value and a and b to 0. Finally, b is read and, since it is now 0, the earlier sensitive value of c is used in the final read.

It is precisely such vulnerabilities that this paper aims to detect. Weak memory reasoning or speculative execution analysis alone would not reveal the issue.

3 Background

Our information flow logic for detecting Spectre-style vulnerabilities builds on the existing information flow logic of Winter et al. [43]. That logic introduces *proof obligations* during weakest precondition (*wp*) reasoning [16,17] to detect insecure information flow: the failure of such a proof obligation implies it is possible for sensitive information to leak to a variable accessible by an attacker.

The logic is sound with respect to the standard notion of *non-interference* [19] where the values of variables with a particular security classification are not influenced by the values of those with higher classifications. Hence, an attacker who can observe the former cannot deduce anything about the latter. This has been demonstrated in Isabelle/HOL over a programming language introduced in [43] with an extension to support simple array operations.

The logic also supports *value-dependent security policies* which enable a variable's security classification to change as the program executes [26,30,32], and thread-local analysis of concurrent code using rely/guarantee reasoning [23,44]. It does not, however, support reasoning on weak memory models. Hence, in this paper we also employ the notion of *reordering interference freedom* (*rif*) checks [14] to detect additional vulnerabilities due to weak memory. As detailed in [14,15], *rif* is readily customised to different processor architectures. It has also been automated and shown to be sound both on a simple while language and an abstraction of ARMv8 assembly code using Isabelle/HOL.

3.1 Weakest Precondition Based Information Flow Reasoning

As is standard in information flow logics, Winter et al. [43] define the security levels relevant to a program as the elements of a lattice $(L, \sqsubseteq)$ where each pair of elements $a, b \in L$ has a *join*, i.e., least upper bound, denoted by $a \sqcup b$, and a *meet*, i.e., greatest lower bound, denoted by $a \sqcap b$. The top of the lattice $\top$ represents the highest security classification, and the bottom $\bot$ the lowest.

A weakest precondition logic traditionally captures, at each point in a program, the weakest predicate needed to maintain correctness from that point in the code. To additionally capture information flow, the wp_{if} logic defined in [43] includes for each variable v an additional variable Γ_v (of type L) denoting the security level of the information currently held by the variable.

Variables are partitioned into *global* variables, which can be accessed by more than one thread, and *local* variables which cannot. In a secure program, for all global variables the security level of the information it holds is less than the variable's security classification. The latter, denoted $\mathcal{L}(v)$, is a conditional expression that evaluates to a value of type L depending on the current program state, i.e., its classification may depend on other variables referred to in this context as *control variables*. Formally, a global variable in a secure program always satisfies $\Gamma_v \sqsubseteq \mathcal{L}(v)$. That is, variables never hold information at a higher security level than their classification, ensuring non-interference. Local variables are not accessible by an attacker, and thus may hold information at any security level. Hence, their security classification is by default the top of the lattice.

When checking whether a particular line of code can leak information, it is assumed that the program is secure up to that point. Hence, for global variables $\Gamma_x \sqcap \mathcal{L}(x)$ is used to denote the security level of the information in variable x. When it is not possible to deduce Γ_x from a program's code, e.g., when x has been assigned to an input, the meet in this expression ensures that its value will not exceed $\mathcal{L}(x)$.

The security level of an expression e in terms of local variables and literals, denoted $\Gamma_E(e)$, is defined as the join of the security levels of the variables to which e refers. That is, $\Gamma_E(e) = \bigsqcup_{r \in vars(e)} \Gamma_r$ where $vars(e)$ denotes the variables occurring in e.

As an example, the wp_{if} rule for an assignment to a global variable $x := e$ replaces each occurrence of variable x with expression e in the post-state Q, and each occurrence of Γ_x with $\Gamma_E(e)$ (denoted $Q[x, \Gamma_x \backslash e, \Gamma_E(e)]$). Additionally, a proof obligation is added to Q to ensure that this change does not violate non-interference. This amounts to checking that

(i) the security level of e is not higher than the security classification of x, and
(ii) since x's value may affect the security classification of other global variables, for each such variable y, y's current security level Γ_y does not exceed its updated security classification with e in place of x.

$$\begin{aligned} wp_{if}(x := e, Q) = {} & Q[x, \Gamma_x \backslash e, \Gamma_E(e)] \wedge \Gamma_E(e) \sqsubseteq \mathcal{L}(x) \wedge \\ & (\forall\, y \cdot \Gamma_y \sqcap \mathcal{L}(y) \sqsubseteq \mathcal{L}(y)[x \backslash e]) \end{aligned}$$

Note that if y is not dependent on x, then $\mathcal{L}(y)[x\backslash e]$ simplifies to $\mathcal{L}(y)$ making the final proof obligation trivially true.

To analyse a thread within this framework, we would start with the predicate *true* holding after the program and step backwards through the code, transforming the predicate (from Q to Q') with $Q' = wp_{if}(\alpha, Q)$ for each instruction α. A *true* postcondition is used since we are focussed on information flow security (whose proof obligations are introduced by the logic) and not functional correctness (which would require a postcondition). The proof obligations added by the wp_{if} transformer at each step have been proven to ensure non-interference, i.e., that sensitive information is not leaked [43].

3.2 Extending with Rely/Guarantee Reasoning

To support reasoning about threads in a concurrent program, the wp_{if} logic is extended with rely/guarantee reasoning [23,44]. Each thread has a rely condition $\mathcal{R}$ and guarantee condition $\mathcal{G}$. The rely condition is a reflexive and transitive relation on states that abstractly captures changes that the environment may make to global variables. The guarantee condition is a reflexive relation on states that abstractly captures changes to global variables that the thread itself is allowed to make. Both conditions are expressed in terms of global variables x and x', the latter representing the variable in the post-state of the relation.

For each instruction α which updates global variables, the corresponding wp_{if} rule is updated to include a proof obligation $guar(\mathcal{G}, \alpha)$ which captures the conditions under which executing α will ensure $\mathcal{G}$. Additionally, all rules are updated with a proof obligation that their other proof obligations are stable, i.e., cannot be falsified, under $\mathcal{R}$. Given P comprises $wp_{if}(\alpha, Q)$ and $guar(\mathcal{G}, \alpha)$ if applicable, stability of P is defined as $stable_{\mathcal{R}}(P) = P \wedge (\forall\, glb' \cdot \mathcal{R} \Rightarrow P')$ where glb' is the list of post-state global variables, and P' is the predicate P with all global variables x replaced by x'. The resulting logic is referred to as wp_{if}^{RG}.

3.3 Reordering Interference Freedom

To take into account instruction reordering due to executing on a weak memory model it is sufficient to check that such reordering cannot invalidate the logic's outcome for a particular program. To do this, we employ the *reordering interference freedom* (*rif*) approach of Coughlin et al. [14]. This approach covers most modern processor architectures (such as x86 and ARMv8) and can be extended to cover all others as shown in [15]. Essentially the approach checks, for every pair of reorderable instructions, α and β, that executing the instructions in the reverse order does not introduce new behaviour. Reorderable instructions are defined in terms of the specific hardware memory model, e.g., TSO, ARMv8, based on the approach of Colvin and Smith [10].

For instructions α and β, let $\beta' \prec \alpha \prec \beta$ be the predicate that β may reorder before α where it is executed as β' (changes to β are due to the possibility of forwarding values from a later write instruction to an earlier read [10]). Given the rely and guarantee conditions under which α and β execute, we define

$$\mathit{rif}_{\mathsf{a}}(\alpha, \beta) \mathrel{\widehat{=}} \forall\, Q \cdot \mathit{wp}_{\mathit{if}}^{RG}(\alpha;\beta, Q) \Rightarrow \mathit{wp}_{\mathit{if}}^{RG}(\beta';\alpha, Q)$$

which expresses that the order of execution of α and β does not affect the security of their execution. This extends to programs p such that

$$\mathit{rif}(p) \ \mathrel{\widehat{=}}\ \forall\, \alpha, \beta \in p \cdot (\beta' \prec \alpha \prec \beta) \Rightarrow \mathit{rif}_{\mathsf{a}}(\alpha, \beta).$$

The *rif* approach is sound because it is defined over all possible post-states Q. Hence, all traces (arising from different sequences of reorderings) under which a reordering could occur are taken into account.

The approach separates the inter-thread interference (using rely/guarantee) from the intra-thread (reordering) interference (using *rif*). That is, *rif* is thread-local. For a thread with n instructions, the worst case is that every instruction can reorder giving us $n(n-1)/2$ reorderable pairs (significantly less than the $n!$ traces that such reordering would introduce). Note also that this worst case is extremely unlikely. Instructions which refer to the same variable are not generally reorderable.

Pairs of potentially reorderable instructions can be identified via a dataflow analysis, similar to dependence analysis commonly used in compiler optimisation. We have previously provided such an automation for both a simple while language and an abstraction of ARMv8 assembly code using Isabelle/HOL [14].

4 Information Flow Logic

Our approach to extend the logic of Winter et al. [43] with speculation is to develop a weakest precondition transformer, wp_s, which operates over *pairs* of predicates $\langle Q_s, Q\rangle$. The predicate Q_s (resp. Q) represents the weakest precondition at that point in the program, assuming the processor is speculating (resp. is not speculating).

Furthermore, proof obligations within Q_s must distinguish between two versions of each global variable: *base* variables (those in the global state visible to all threads), and *frame* variables (those in the local speculation frame of this thread only). While speculating, writes and subsequent reads of global variables will only access the frame variables. Reads of global variables without a previous write during speculation will access the base variables. Other threads will concurrently read and write to the base variables. Within Q_s, we denote base variables with a $\flat$ superscript so that predicates in terms of them will not be transformed by wp_s over speculative instructions.

Conceptually, wp_s can be understood as two wp transformers, one for the speculative case and one for the non-speculative case, running in parallel over each instruction. For a program to be secure, its precondition must imply the non-speculative weakest precondition. The speculative weakest precondition is merged into the speculative one at each branching point. This framework enables reasoning about speculation, even nested speculation, in a manner very similar to ordinary *wp* reasoning and with minimal added complexity.

4.1 Weakest Precondition with Speculation

For ease of presentation, we define wp_s over the instructions of a high-level programming language representing assembly programs (as in [10]). The syntax of an instruction, α, and a program, p, is defined as follows.

$$\begin{aligned}\alpha &::= \mathsf{skip} \mid r := e \mid r := x \mid x := e \mid \mathsf{fence} \mid \mathsf{leak}\ e\\ p &::= \alpha \mid p \mathbin{;} p \mid \mathsf{if}\ b\ \mathsf{then}\ p\ \mathsf{else}\ p \mid \mathsf{while}\ b\ \mathsf{do}\ p\end{aligned}$$

where r is a local variable, x is a global variable, b a Boolean condition and e an expression. Both b and e are in terms of local variables and literals only, reflecting the use of registers for these values in assembly code. The language includes a fence instruction which prevents reordering of instructions and also terminates current speculative execution. A special *ghost* instruction[1] leak e is inserted into a program to indicate that the following instructions are a gadget that leaks information through a micro-architectural side channel when executed (speculatively, or otherwise).

Before analysing a program with our logic, we insert leak instructions before each gadget of interest during a pre-pass over the code. Since typical gadgets can be detected syntactically, this is a straightforward task to mechanise. The expression e of the inserted leak instruction is based on what information leaks when the gadget is used in an attack. For the example of Section 2.2 where the memory access [r2] would leak r2, e would be $r2$. After this pre-pass the code is analysed using our logic to determine whether the information leaked is possibly sensitive and hence the gadget causes a security vulnerability. Note that not all code conforming to the syntactic form of a gadget will enable a successful attack on sensitive information.

Since the pre-pass can be customised for different gadgets, the overall approach can be adapted to a variety of attacks, including new attacks as they are discovered. We discuss adapting the approach for a number of existing speculative execution attacks in Section 4.5.

The rules for our speculative execution logic wp_s build on those of wp_{if} [43]. We extend them with rely/guarantee reasoning in Section 4.2. A formal proof in Isabelle/HOL of the soundness of the resulting rules with respect to an abstract semantics of speculative execution [11] is available online [12].

Skip: A skip instruction does not change the $\langle Q_s, Q\rangle$ tuple.

$$wp_s(\mathsf{skip}, \langle Q_s, Q\rangle) = \langle Q_s, Q\rangle$$

Local assignment: Local variables may hold information at any security level and cannot be used as control variables. Hence, we do not need the proof obligations

[1] A ghost instruction is not part of the actual code and is used for analysis purposes only.

of global assignments detailed in Section 3.1. For assigning an expression e to a local variable, we have

$$wp_s(r := e, \langle Q_s, Q\rangle) = \langle Q_s[r, \Gamma_r \backslash e, \Gamma_E(e)], Q[r, \Gamma_r \backslash e, \Gamma_E(e)]\rangle\,.$$

For assigning the value of a global variable x to a local variable, $r := x$, we need to modify the weakest precondition in the speculative state. Since we do not know whether a load of x during speculation is of a frame variable or a base variable (which is subject to interference from other threads) we need to consider both cases. Let *glb* be the list of *globals*, i.e., all global variables x and their associated Γ_x variables. For the case where the load is of the base variable, we replace each $y \in glb$ with $y^\flat$. This ensures the predicate is not transformed by speculative assignments as we reason backwards through the code. It is only transformed by the assignments of other threads via the rely condition (as will be described in Section 4.2), and so remains consistent with the actual values and classifications of globals in terms of the base variables (which are shared with other threads).

To ensure in the reasoning that the correct case is used, we distinguish the cases by qualifying them with whether x has been written to during the speculation and hence is defined in the frame, or not. To do this, we introduce a ghost variable x_{def} which is true when x is defined within the frame, and false otherwise. When x is defined by an earlier write during speculation (and hence the later load was from the frame) then x_{def} will be set to true (see the global assignment rule below). This will cause the base case to be ignored, leaving just the frame case.

If, on the other hand, there is no such earlier write to x, both cases reach the start of speculation where x_{def} will be set to false (see the if and while rules below). This will cause the frame case to be ignored, leaving just the base case.

Formally, we have

$$\begin{aligned} wp_s(r := x, \langle Q_s, Q\rangle) = \langle &(x_{def} \Rightarrow Q_s[r, \Gamma_r \backslash x, \Gamma_x]) \wedge \\ &(\neg x_{def} \Rightarrow Q_s[r, \Gamma_r \backslash x^\flat, \Gamma_x^\flat \sqcap \mathcal{L}(x)[glb \backslash glb^\flat]]), \\ &Q[r, \Gamma_r \backslash x, \Gamma_x \sqcap \mathcal{L}(x)]\rangle\,. \end{aligned}$$

where $glb^\flat$ is the list *glb* with each element y replaced by $y^\flat$. Note that in the base case of the speculative precondition all globals y are replaced by $y^\flat$. This ensures that they refer to the values of the base variables. In the frame case, on the other hand, x and Γ_x refer to the frame variables and will be transformed by wp_s over earlier speculative assignments.

Global assignment: An assignment to a global variable $x := e$ sets x_{def} to true and replaces each occurrence of variable x and Γ_x with expression e and security level $\Gamma_E(e)$, respectively, in both Q_s and Q. Additionally, in the non-speculative case we have the proof obligations of wp_{if} described in Section 3.1. The speculative case does not have these proof obligations. Since a speculatively executed assignment does not write to memory, it has no effect on the classification of other variables.

$$\begin{aligned} wp_s(x := e, \langle Q_s, Q\rangle) = \langle & Q_s[x, \Gamma_x, x_{def} \backslash e, \Gamma_E(e), true], \\ & Q[x, \Gamma_x \backslash e, \Gamma_E(e)] \wedge \Gamma_E(e) \sqsubseteq \mathcal{L}(x) \wedge \\ & (\forall\, y \cdot \Gamma_y \sqcap \mathcal{L}(y) \sqsubseteq \mathcal{L}(y)[x \backslash e])\rangle \end{aligned}$$

Fence: The fence instruction terminates any current speculative execution. Hence, any proof obligations in the speculative state beyond the fence do not need to be considered at the point in the program where a fence occurs. Q_s is therefore replaced by *true* and Q is unchanged.

$$wp_s(\mathsf{fence}, \langle Q_s, Q\rangle) = \langle true, Q\rangle$$

Leak: The instruction leak e leaks the value of expression e via a micro-architectural side channel, introducing a proof obligation into both Q_s and Q.

$$wp_s(\mathsf{leak}\ e, \langle Q_s, Q\rangle) = \big\langle Q_s \wedge \Gamma_E(e) = \bot,\ Q \wedge \Gamma_E(e) = \bot\big\rangle$$

where $\bot$ denotes the lowest value of the security lattice. Requiring that the leaked information is at this level ensures that the attacker cannot deduce anything new from the information, regardless of the level of information they can observe.
Sequential composition: As in standard *wp* reasoning, sequentially composed instructions transform the tuple one at a time.

$$wp_s(p_1\, ;\, p_2, \langle Q_s, Q\rangle) = wp_s(p_1, wp_s(p_2, \langle Q_s, Q\rangle))$$

If-then-else: In general, an if statement might occur within a speculative context (when nested in or following an earlier if, for example). The branch that is followed speculatively is, in general, independent of that actually executed later. Hence, the speculative proof obligations from both branches are conjoined to form the speculative precondition.

Additionally, given that the if statement might initiate speculation, the speculative proof obligations need to be merged into the non-speculative precondition. We do this by (i) setting x_{def} for all global variables x to false in the speculative precondition, leaving just the base case, and then (ii) renaming each global $y^\flat$ to y so that the resulting speculative precondition Q_s can be conjoined with the non-speculative precondition Q.

Finally, a proof obligation $\Gamma_E(b) = \bot$ is added to the non-speculative precondition. Such a proof obligation is common in information flow logics for concurrent programs since the value of b can readily be deduced using timing attacks on such programs [31,37]. It is not necessary to also check this proof obligation in the speculative case whose purpose is to detect vulnerabilities that are *not* detectable in the non-speculative case.

With $\langle Q_{s1}, Q_1\rangle = wp_s(p_1, \langle Q_s, Q\rangle)$ and $\langle Q_{s2}, Q_2\rangle = wp_s(p_2, \langle Q_s, Q\rangle)$, we have

$$\begin{aligned} & wp_s(\mathsf{if}\ b\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2, \langle Q_s, Q\rangle) = \\ & \quad \langle\, Q_{s1} \wedge Q_{s2}, \Gamma_E(b) = \bot \wedge (b \Rightarrow Q_1) \wedge (\neg\, b \Rightarrow Q_2) \wedge \\ & \qquad\qquad (Q_{s1} \wedge Q_{s2})[glb^\flat, d_1, ..., d_n \backslash glb, false, ..., false]\,\rangle\,. \end{aligned}$$

where $glb^\flat$ is the list glb with all elements y replaced by $y^\flat$, and $d_1, .., d_n$ is the list of introduced ghost variables of the form x_{def}.

While-do: Similar to standard *wp* reasoning, we can soundly approximate the weakest precondition of a loop by finding invariants which imply our speculative and non-speculative postconditions, Q_s and (when the loop guard is false) Q, and which are maintained by the loop body (when the loop guard is true in the non-speculative case). As with the if rule, a proof obligation $\Gamma_E(b) = \bot$ must hold in the non-speculative case.

$$wp_s(\mathsf{while}\ b\ \mathsf{do}\ p, \langle Q_s, Q\rangle) = \langle Inv_s, Inv\rangle$$

where $Inv_s \Rightarrow Q_s$ and $Inv \Rightarrow \Gamma_E(b) = \bot \wedge Inv_s[glb^\flat, d_1, ..., d_n \backslash glb, false, ..., false]$ and $Inv \wedge \neg\, b \Rightarrow Q$, and given $wp_s(p, \langle Inv_s, Inv\rangle) = \langle P_s, P\rangle$, then $Inv_s \Rightarrow P_s$ and $Inv \wedge b \Rightarrow P$. Like the if rule, the while rule copies the proof obligations in the speculative precondition to the non-speculative precondition, and maintains those in the speculative precondition in case the loop is reached within an existing speculative context.

4.2 Rely/Guarantee and Reordering

Given $\langle P_s, P\rangle = wp_s(\alpha, \langle Q_s, Q\rangle)$, we account for a thread's rely and guarantee conditions, $\mathcal{R}$ and $\mathcal{G}$, by ensuring that P_s and P are *stable*, i.e., cannot be made false under changes allowed by $\mathcal{R}$, and that α's effects on global variables satisfy $\mathcal{G}$.

For P_s, frame variables y will be unaffected by the environment whereas base variables $y^\flat$ represent the actual globals and will be subject to environmental change. P_s, therefore, needs to be stable under $id_{glb} \wedge \mathcal{R}[glb\backslash glb^\flat]$ where id_{glb} equates each $y \in glb$ with y', and $glb^\flat$ is the list glb with all elements y renamed to $y^\flat$.

The guarantee $\mathcal{G}$ need only be considered for global assignments $x := e$ as these are the only instructions that affect the shared environment. For such assignments, we require that $\mathcal{G}$ holds in P when e is used in place of x', and y is used in place of y' for all other variables, i.e., these variables are unchanged by the assignment. This is not needed for P_s, as globals are unchanged when executing speculatively.

Given glb' (resp. $glb^{\flat\prime}$) is the list glb with each element y replaced by y' (resp. $y^{\flat\prime}$), and $\langle P_s, P\rangle = wp_s(\alpha, \langle Q_s, Q\rangle)$, we define wp_s^{RG} for instructions as

$$\begin{aligned} wp_s^{RG}(\alpha, \langle Q_s, Q\rangle) = \langle\, & P_s \wedge (\forall\, glb', glb^{\flat\prime} \cdot id_{glb} \wedge \mathcal{R}[glb\backslash glb^\flat] \Rightarrow P_s'), \\ & P \wedge \mathcal{G}^\alpha \wedge (\forall\, glb' \cdot \mathcal{R} \Rightarrow (P \wedge \mathcal{G}^\alpha)[glb\backslash glb'])\,\rangle \end{aligned}$$

where $\mathcal{G}^\alpha$ is defined as $\mathcal{G}[x'\backslash e][glb'\backslash glb]$ when α is $x := e$ and as *true* for all other instructions.

For program structures, e.g., sequential composition, wp_s^{RG} is defined equivalently to wp_s, with all recursive invocations replaced with wp_s^{RG} and all loop invariants stable under $\mathcal{R}$.

4.3 Reordering Interference Freedom

Once a thread of our program has been proven secure with the logic wp_s^{RG}, we separately check reordering interference freedom (*rif*). This will uncover any problems due to reordering interference such as that in the example of Section 2.2. For reasoning over state tuples of the logic wp_s^{RG}, we define rif_{a} as

$$rif_{\mathsf{a}}(\alpha, \beta) = \forall\, Q_s, Q \cdot wp_s^{RG}(\alpha;\beta, \langle Q_s, Q\rangle) \Rightarrow wp_s^{RG}(\beta';\alpha, \langle Q_s, Q\rangle)$$

where $\langle Q_{s1}, Q_1\rangle \Rightarrow \langle Q_{s2}, Q_2\rangle$ is defined to be $(Q_{s1} \Rightarrow Q_{s2}) \wedge (Q_1 \Rightarrow Q_2)$.

4.4 Example Revisited

Applying wp_s^{RG} to the example of Section 2.2 results in a weakest precondition of true, revealing no security vulnerability as expected when weak memory is not taken into account. A *rif* check, however, reveals that the reordering of the syntactically independent instructions `r0 := b` and `r1 := c` can lead to different behaviour, indicating that the program *may* be insecure.

We investigate this possibility by applying wp_s^{RG} to the example with the instructions reordered in Fig. 1. We customise Γ_E so that $\Gamma_E(r\hat{}r) = \bot$ given that the result of this expression will always be 0, and $\Gamma_E(e\ ?\ t : f) = \Gamma_E(e) \sqcup$ (if e then $\Gamma_E(t)$ else $\Gamma_E(f)$) to reflect that the security level of the expression will depend on just one of t or f. To improve precision, such customisations would be built into the logic for expressions in the given programming language to which it is applied.

We let $\mathcal{R} = (a = 0 \Rightarrow a' = 0) \wedge (b = 0 \Rightarrow b' = 0)$ to capture that once either a or b is zero it never changes. $\mathcal{L}(a)$ and $\mathcal{L}(b)$ are $\bot$ (the lowest security level) in any state, and $\mathcal{L}(c)$ is $\bot$ whenever $a = 0 \vee b = 0$. Where two predicate pairs appear between lines of code, the upper one is a simplification of the lower. Since there are no writes to global variables in the branch, for presentation purposes we have only included those predicates in the speculative states corresponding to all global variables being identified with the base variables (other predicates are replaced with ...).

None of the instructions change global variables, and hence there are no guarantee checks. Stability checks are required, however, for those predicates in terms of globals. The conjunct $\Gamma_{r1} = \bot$ above the line `r0 := b` is introduced as there are no states in which the predicate $b \neq 0$ is stable, i.e., $b = 0 \Rightarrow \Gamma_{r1} = \bot$ is stable only when $\Gamma_{r1} = \bot$.

Since a and b do not change when they are 0, and c does not hold sensitive information when a or b are 0, the predicate $\Gamma_c = \bot \vee a = 0 \vee b = 0$ (above the line `r1 := c`) is stable when $a = 0 \vee b = 0$. Similarly, for the speculative predicate $\Gamma_c^\flat = \bot \vee a^\flat = 0 \vee b^\flat = 0$. Also, the predicate $r0 = r1 \Rightarrow a = 0 \vee b = 0$ (above the line `if (r0=r1)`) is stable when $a = 0 \vee b = 0$.

The calculated weakest precondition at the beginning of the example code is $a = 0 \vee b = 0$, indicating that the code is only secure when both locks are not held. Therefore, the checks of a and b do not have the effect that the programmer

```
$\langle ..., a = 0 \vee b = 0\rangle$
r0 := a;
$\langle ..., \Gamma_{r0} = \bot \wedge (a = 0 \vee b = 0)\rangle$
r1:= r1^r1;
$\langle ..., \Gamma_{r0} \sqcup \Gamma_{r1} = \bot \wedge (a = 0 \vee b = 0)\rangle$
$\langle ..., \Gamma_E(r0 = r1) = \bot \wedge (r0 = r1 \Rightarrow a = 0 \vee b = 0) \wedge (a = 0 \vee b = 0)\rangle$
if (r0 = r1)
  $\langle ... \wedge (\neg b_{def} \wedge \neg c_{def} \Rightarrow a^\flat = 0 \vee b^\flat = 0), a = 0 \vee b = 0\rangle$
  $\langle ... \wedge (\neg b_{def} \wedge \neg c_{def} \Rightarrow P[glb \backslash glb^\flat]), P\rangle$
    where $P \mathrel{\widehat{=}} (\Gamma_c = \bot \vee a = 0 \vee b = 0) \wedge (a = 0 \vee b = 0)$
  r1 := c;
  $\langle ... \wedge (\neg b_{def} \Rightarrow \Gamma_{r1} = \bot), \Gamma_{r1} = \bot\rangle$
  $\langle ... \wedge (\neg b_{def} \Rightarrow P[glb \backslash glb^\flat]), P\rangle$
    where $P \mathrel{\widehat{=}} \Gamma_b \sqcap \mathcal{L}(b) = \bot \wedge (b = 0 \Rightarrow \Gamma_{r1} = \bot) \wedge \Gamma_{r1} = \bot$
  r0 := b;
  $\langle \Gamma_{r0} = \bot \wedge (r0 = 0 \Rightarrow \Gamma_{r1} = \bot), \Gamma_{r0} = \bot \wedge (r0 = 0 \Rightarrow \Gamma_{r1} = \bot)\rangle$
  $\langle \Gamma_E((r0 = 0)\,?\,base + r1 : 0)) = \bot, \Gamma_E((r0 = 0)\,?\,base + r1 : 0) = \bot\rangle$
  r2 := (r0 = 0) ? base + r1 : 0;
  $\langle \Gamma_{r2} = \bot, \Gamma_{r2} = \bot\rangle$
  leak(r2);
  $\langle true, true\rangle$
  r3 := [r2]
  $\langle true, true\rangle$
else
  $\langle true, true\rangle$
  skip
  $\langle true, true\rangle$          where $glb^\flat$ is the list $glb$ with all elements $y$ replaced by $y^\flat$.
```

Fig. 1. Applying wp_s^{RG} to a reordering of the example from Section 2.2.

intended. The check on b fails due to the reordered loads on c and b, whereas the check on a fails due to the incorrect speculation of the branch `if (r0 = r1)`. The latter is evident in its precondition constraining $a = 0 \vee b = 0$ regardless of the branch condition. Since not taking into account either instruction reordering or speculation would be sufficient to establish $a = 0 \vee b = 0$ (resulting in the weakest precondition being true), this vulnerability can only be identified by considering both.

4.5 Discussion

The use of leak instructions in our logic means we can readily customise it to detect various speculative execution vulnerabilities. For example, to detect Spectre variant 1 vulnerabilities, we would extend the logic with arrays (as was done for the Isabelle/HOL encoding of wp_{if} [43]). The expression e of the leak instruction would be the value used for the final read in the standard gadget (see Section 2.1). This simple extension of the logic would also allow us to detect (i) BranchSpec vulnerabilities [9] where a sensitive value from a speculative array-out-of-bounds access is used as a branching condition, and (ii) vulnerabilities to the data variant of the PACMAN attack [34]. During speculation, this attack obtains a sensitive value related to a Pointer Authentication Code (PAC), a recent security feature of ARM processors. It then performs a load using this sensitive value as the address to make the value accessible to the attacker (via a cache timing attack) after the speculation.

On the other hand, the instruction variant of PACMAN relies on a speculatively executed indirect branch to a sensitive address. Indirect branches could also be incorporated as in the weakest precondition calculus in [2]. A proof obligation that such a branch is only taken on values with security level $\perp$ could then be added to the speculative precondition of the branch.

Ren et al. [35] describe two gadgets relying on a sensitive value accessed during speculation being leaked by the micro-op cache of Intel and AMD processors. These gadgets are based on function calls, and fetches of indirect branches. The latter allows bypassing of a fence intended to stall speculative execution. To detect related vulnerabilities, we would need to further extend our programming language with function calls, and change the fence rule to allow later instructions to be "fetched" but not executed.

As well as Spectre-PHT, the initial paper by Kocher et al. [24] describe Spectre-BTB (or variant 2), which targets the branch target buffer used by the processor to predict destinations of indirect branch instructions. Compared to Spectre-PHT, this is more powerful. Any indirect branch is potentially vulnerable, and an attacker's mistraining can direct speculation towards a convenient gadget anywhere in the program or library code. Detecting such gadgets may still be possible in our approach if used in conjunction with recent compiler-based mitigations, Serbeus [29] or Switchpoline [3], which vastly reduce the potential target addresses of indirect branch instructions. This also applies to other approaches based on a speculative indirect branch (call, jump or return) to a gadget, such as SMoTherSpectre [4] and RETBLEED [42].

5 Related Work

Cauligi et al. [8] provide a comprehensive overview of existing semantics and tools aimed at providing formal reasoning about speculative execution. Only 7 of the 24 papers they examine consider out-of-order execution. These either model the mechanism for instruction reordering directly (in terms of a multi-stage *fetch-execute-retire* pipeline) [7,20,21,40], or capture the effects of instruction reordering via higher level abstractions: reordering relations [11], pomsets [18] and event graphs [33].

The former provide more precise characterisations of the hardware and hence the potential to detect a wider variety of vulnerabilities than more abstract approaches. However, such detailed models also add complexity to the verification task. For this reason, all of these models support analysis on only a single thread, and hence are unable to detect the kinds of leakage illustrated by our running example from Section 2.2.

The abstraction-based approaches of Disselkoen et al. (based on pomsets) [18] and Ponce de León and Kinder (based on event graphs) [33], use intra- and inter-thread relations between instructions to capture instruction ordering in concurrent programs (unrelated instructions can be reordered). The inter-thread relations are necessary for these approaches, but preclude thread-local reasoning. Instead, reasoning is over individual executions of a full program.

Our approach builds directly on the abstract semantics of Colvin and Winter [11]. That paper introduces the idea of a speculative context that operates on a fresh copy of the program state, which is key to our approach. It models out-of-order execution via a relation capturing which pairs of instructions in a given thread can reorder. Since the relation only imposes intra-thread constraints, the semantics can be used in a thread-local analysis, avoiding analysis over the exponential explosion of behaviours possible due to interleaving in a full concurrent program. The *rif* approach we adopt from [14] also uses such a reordering relation enabling our thread-local approach.

We extend the semantics of [11] with an information flow logic which defines the capabilities of an attacker in terms of which parts of memory they can observe, and their ability to observe control flow (via timing). The latter, in particular, is listed as an open problem by Cauligi et al. [8] for semantics based on abstractions of out-of-order execution.

6 Conclusion

In this paper, we have shown how information leakage can occur due to a combination of speculative execution and out-of-order execution, both of which are features of modern processors. To the best of our knowledge, this is the first paper to demonstrate that such a leak is possible. To enable detection of such leaks, we have developed a novel information flow logic using weakest precondition reasoning over a tuple of states comprising the actual and speculative states of the program. For scalability, the logic supports thread-local reasoning, and a notion of reordering interference freedom (*rif*), both of which significantly reduce the number of behaviours that must be analysed: the former allows us to abstract from concurrent interleaving of threads in a program, and the latter allows us to replace reasoning over behaviours resulting from instruction reordering by pair-wise checks over reorderable instructions. Our logic has been proven sound with respect to an abstract semantics of speculative execution [11] using Isabelle/HOL [12].

Our future goals include mechanising the information flow logic in the auto-active program verifier Boogie [1]. This will build on an existing encoding of information flow and rely/guarantee reasoning in Dafny [36], and require a way to support our novel representation of program state as a tuple of speculative and actual state spaces.

Acknowledgments. We would like to thank the anonymous referees for their insightful comments and suggestions.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005. Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer (2005). https://doi.org/10.1007/11804192_17
2. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: Ernst, M.D., Jensen, T.P. (eds.) Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'05. pp. 82–87. ACM (2005). https://doi.org/10.1145/1108792.1108813
3. Bauer, M., Hetterich, L., Rossow, C., Schwarz, M.: Switchpoline: A software mitigation for Spectre-BTB and Spectre-BHB on ARMv8. In: 2024 ACM ASIA Conference on Computer and Communications Security, AsiaCCS 2024. ACM (2024). https://doi.org/https://doi.org/10.60882/cispa.25304857.v1
4. Bhattacharyya, A., Sandulescu, A., Neugschwandtner, M., Sorniotti, A., Falsafi, B., Payer, M., Kurmus, A.: SMoTherSpectre: Exploiting speculative execution through port contention. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019. pp. 785–800. ACM (2019). https://doi.org/10.1145/3319535.3363194
5. Bulck, J.V., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y., Strackx, R.: Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In: Enck, W., Felt, A.P. (eds.) 27th USENIX Security Symposium, USENIX Security 2018. pp. 991–1008. USENIX Association (2018)
6. Canella, C., Bulck, J.V., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., Gruss, D.: A systematic evaluation of transient execution attacks and defenses. In: Heninger, N., Traynor, P. (eds.) 28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019. pp. 249–266. USENIX Association (2019)
7. Cauligi, S., Disselkoen, C., von Gleissenthall, K., Tullsen, D.M., Stefan, D., Rezk, T., Barthe, G.: Constant-time foundations for the new Spectre era. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020. pp. 913–926. ACM (2020). https://doi.org/10.1145/3385412.3385970
8. Cauligi, S., Disselkoen, C., Moghimi, D., Barthe, G., Stefan, D.: SoK: Practical foundations for software Spectre defenses. In: 43rd IEEE Symposium on Security and Privacy, SP 2022. pp. 666–680. IEEE (2022). https://doi.org/10.1109/SP46214.2022.9833707
9. Chowdhuryy, M.H.I., Liu, H., Yao, F.: Branchspec: Information leakage attacks exploiting speculative branch instruction executions. In: 38th IEEE International Conference on Computer Design, ICCD 2020. pp. 529–536. IEEE (2020). https://doi.org/10.1109/ICCD50377.2020.00095
10. Colvin, R.J., Smith, G.: A wide-spectrum language for verification of programs on weak memory models. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E.P. (eds.) Formal Methods - 22nd International Symposium, FM 2018. Lecture Notes in Computer Science, vol. 10951, pp. 240–257. Springer (2018). https://doi.org/10.1007/978-3-319-95582-7_14

11. Colvin, R.J., Winter, K.: An abstract semantics of speculative execution for reasoning about security vulnerabilities. In: Sekerinski, E., et al. (eds.) Formal Methods. FM 2019 International Workshops - Revised Selected Papers, Part II. Lecture Notes in Computer Science, vol. 12233, pp. 323–341. Springer (2019). https://doi.org/10.1007/978-3-030-54997-8_21
12. Coughlin, N., Lam, K., Winter, K.: Weak memory rely/guarantee logic with speculative execution. https://github.com/UQ-PAC/wmm-rg/tree/paperwork-st (April 2024)
13. Coughlin, N., Smith, G.: Compositional noninterference on hardware weak memory models. Sci. Comput. Program. **217**, 102779 (2022). https://doi.org/10.1016/j.scico.2022.102779
14. Coughlin, N., Winter, K., Smith, G.: Rely/guarantee reasoning for multicopy atomic weak memory models. In: Huisman, M., Pasareanu, C.S., Zhan, N. (eds.) Formal Methods - 24th International Symposium, FM 2021. Lecture Notes in Computer Science, vol. 13047, pp. 292–310. Springer (2021). https://doi.org/10.1007/978-3-030-90870-6_16
15. Coughlin, N., Winter, K., Smith, G.: Compositional reasoning for non-multicopy atomic architectures. Formal Aspects Comput. **35**(2), 8:1–8:30 (2023). https://doi.org/10.1145/3574137
16. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976), https://www.worldcat.org/oclc/01958445
17. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Springer-Verlag, Berlin, Heidelberg (1990)
18. Disselkoen, C., Jagadeesan, R., Jeffrey, A., Riely, J.: The code that never ran: Modeling attacks on speculative evaluation. In: 2019 IEEE Symposium on Security and Privacy, SP 2019. pp. 1238–1255. IEEE (2019). https://doi.org/10.1109/SP.2019.00047
19. Goguen, J.A., Meseguer, J.: Security policies and security models. In: 1982 IEEE Symposium on Security and Privacy, 1982. pp. 11–20. IEEE Computer Society (1982). https://doi.org/10.1109/SP.1982.10014
20. Guanciale, R., Balliu, M., Dam, M.: InSpectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security. pp. 1853–1869. ACM (2020). https://doi.org/10.1145/3372297.3417246
21. Guarnieri, M., Köpf, B., Reineke, J., Vila, P.: Hardware-software contracts for secure speculation. In: 42nd IEEE Symposium on Security and Privacy, SP 2021. pp. 1868–1883. IEEE (2021). https://doi.org/10.1109/SP40001.2021.00036
22. Islam, S., Moghimi, A., Bruhns, I., Krebbel, M., Gülmezoglu, B., Eisenbarth, T., Sunar, B.: SPOILER: speculative load hazards boost Rowhammer and cache attacks. In: Heninger, N., Traynor, P. (eds.) 28th USENIX Security Symposium, USENIX Security 2019. pp. 621–637. USENIX Association (2019)
23. Jones, C.B.: Specification and design of (parallel) programs. In: IFIP Congress. pp. 321–332 (1983)
24. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. In: 2019 IEEE Symposium on Security and Privacy, SP 2019. pp. 1–19. IEEE (2019). https://doi.org/10.1109/SP.2019.00002
25. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading

kernel memory from user space. In: Enck, W., Felt, A.P. (eds.) 27th USENIX Security Symposium, USENIX Security 2018. pp. 973–990. USENIX Association (2018)

26. Lourenço, L., Caires, L.: Dependent information flow types. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015. pp. 317–328. ACM (2015). https://doi.org/10.1145/2676726.2676994
27. Mantel, H., Perner, M., Sauer, J.: Noninterference under weak memory models. In: IEEE 27th Computer Security Foundations Symposium, CSF 2014. pp. 80–94. IEEE Computer Society (2014). https://doi.org/10.1109/CSF.2014.14
28. Mantel, H., Sands, D., Sudbrock, H.: Assumptions and guarantees for compositional noninterference. In: Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011. pp. 218–232. IEEE Computer Society (2011). https://doi.org/10.1109/CSF.2011.22
29. Mosier, N., Nemati, H., Mitchell, J.C., Trippel, C.: Serberus: Protecting cryptographic code from spectres at compile-time. In: 2024 IEEE Symposium on Security and Privacy, SP 2024. IEEE (2024). https://doi.org/10.1109/SP54263.2024.00048
30. Murray, T.C.: Short paper: On high-assurance information-flow-secure programming languages. In: Clarkson, M., Jia, L. (eds.) Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security, PLAS@ECOOP 2015. pp. 43–48. ACM (2015). https://doi.org/10.1145/2786558.2786561
31. Murray, T.C., Sison, R., Engelhardt, K.: COVERN: A logic for compositional verification of information flow control. In: 2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018. pp. 16–30. IEEE (2018). https://doi.org/10.1109/EuroSP.2018.00010
32. Murray, T.C., Sison, R., Pierzchalski, E., Rizkallah, C.: Compositional verification and refinement of concurrent value-dependent noninterference. In: IEEE 29th Computer Security Foundations Symposium, CSF 2016. pp. 417–431. IEEE Computer Society (2016). https://doi.org/10.1109/CSF.2016.36
33. Ponce de León, H., Kinder, J.: Cats vs. Spectre: An axiomatic approach to modeling speculative execution attacks. In: 43rd IEEE Symposium on Security and Privacy, SP 2022. pp. 235–248. IEEE (2022). https://doi.org/10.1109/SP46214.2022.9833774
34. Ravichandran, J., Na, W.T., Lang, J., Yan, M.: PACMAN: attacking ARM pointer authentication with speculative execution. IEEE Micro **43**(4), 11–18 (2023). https://doi.org/10.1109/MM.2023.3273189
35. Ren, X., Moody, L., Taram, M., Jordan, M., Tullsen, D.M., Venkat, A.: I see dead μops: Leaking secrets via Intel/AMD micro-op caches. In: 48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021. pp. 361–374. IEEE (2021). https://doi.org/10.1109/ISCA52012.2021.00036
36. Smith, G.: A Dafny-based approach to thread-local information flow analysis. In: 11th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE 2023. pp. 86–96. IEEE (2023). https://doi.org/10.1109/FormaliSE58978.2023.00017
37. Smith, G., Coughlin, N., Murray, T.: Value-dependent information-flow security on weak memory models. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) Formal Methods - The Next 30 Years - Third World Congress, FM 2019. Lecture Notes in Computer Science, vol. 11800, pp. 539–555. Springer (2019). https://doi.org/10.1007/978-3-030-30942-8_32

38. Smith, G., Coughlin, N., Murray, T.: Information-flow control on ARM and POWER multicore processors. Formal Methods Syst. Des. **58**(1-2), 251–293 (2021). `https://doi.org/10.1007/S10703-021-00376-2`
39. Sorin, D.J., Hill, M.D., Wood, D.A.: A Primer on Memory Consistency and Cache Coherence. Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers (2011). `https://doi.org/10.2200/S00346ED1V01Y201104CAC016`
40. Vassena, M., Disselkoen, C., von Gleissenthall, K., Cauligi, S., Kici, R.G., Jhala, R., Tullsen, D.M., Stefan, D.: Automatically eliminating speculative leaks from cryptographic code with Blade. Proc. ACM Program. Lang. **5**(POPL), 1–30 (2021). `https://doi.org/10.1145/3434330`
41. Vaughan, J.A., Millstein, T.D.: Secure information flow for concurrent programs under Total Store Order. In: Chong, S. (ed.) 25th IEEE Computer Security Foundations Symposium, CSF 2012. pp. 19–29. IEEE Computer Society (2012). `https://doi.org/10.1109/CSF.2012.20`
42. Wikner, J., Razavi, K.: RETBLEED: arbitrary speculative code execution with return instructions. In: Butler, K.R.B., Thomas, K. (eds.) 31st USENIX Security Symposium, USENIX Security 2022. pp. 3825–3842. USENIX Association (2022)
43. Winter, K., Coughlin, N., Smith, G.: Backwards-directed information flow analysis for concurrent programs. In: 34th IEEE Computer Security Foundations Symposium, CSF 2021. pp. 1–16. IEEE (2021). `https://doi.org/10.1109/CSF51468.2021.00017`
44. Xu, Q., de Roever, W.P., He, J.: The rely-guarantee method for verifying shared variable concurrent programs. Formal Aspects of Computing **9**(2), 149–174 (1997). `https://doi.org/10.1007/BF01211617`